

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY) 06-10-2004		2. REPORT TYPE Final		3. DATES COVERED (From - To) 27-03-2002 - 27-3-2003	
4. TITLE AND SUBTITLE Programmable Interfaces for Advanced Static Analysis				5a. CONTRACT NUMBER N00014-02-C-0188	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) Tim Teitelbaum				5d. PROJECT NUMBER	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) GrammaTech, Inc.; 317 N. Aurora St., Ithaca, NY 14850				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research 800 North Quincy Street Arlington, VA 22217-5660				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION / AVAILABILITY STATEMENT Unrestricted					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT Tools that analyze software are exorbitant to develop, yet different analysis applications have quite similar infrastructure requirements. An effective cost-reduction approach is to amortize the development costs of a common infrastructure across multiple subject programming languages, computer platforms, and analysis applications. This is the final report of a one-year project aimed at creating such a common program-analysis infrastructure.					
15. SUBJECT TERMS Static analysis infrastructure binary code buffer-overflow model-checking					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT Same as report	18. NUMBER OF PAGES 7	19a. NAME OF RESPONSIBLE PERSON Tim Teitelbaum
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified			19b. TELEPHONE NUMBER (include area code) 607-273-7340

 Standard Form 298 (Rev. 8-98)
 Prescribed by ANSI Std. Z39.18

A Introduction

Tools that analyze software are exorbitant to develop, yet different analysis applications have quite similar infrastructure requirements. An effective cost-reduction approach is to amortize the development costs of a common infrastructure across multiple subject programming languages, computer platforms, and analysis applications. This is the final report of a one-year project aimed at creating such a common program-analysis infrastructure.

B Approach

Our approach to creating an effective common infrastructure was to start with CodeSurfer [2, 3], a tool that was originally narrowly conceived as just a program understanding system for ANSI C, and then adapt it to meet the needs of a collection of representative researchers wanting to use it for other applications. Specifically, we worked with six different efforts within two CIP/SW MURIs at the University of Wisconsin and Carnegie-Mellon University (CMU) managed by the Office of Naval Research (ONR), and four separately funded GrammaTech projects with quite similar goals:

#	Application	A. Wisconsin / CMU Projects	B. GrammaTech Projects
1	Object Code Analysis	Analysis of COTS executables for the Intel x86 family of processors (Reps, Balakrishnan)	Analysis of firmware for the Intel x86 family of processors (AFRL/Rome SBIR Phase-I and Phase-II projects “Detecting Malicious Code in Firmware”)
2	Buffer-Overrun Analysis	Detection of buffer-overflow vulnerabilities in C source code (Jha, Ganapathy)	Detection of buffer-overflow vulnerabilities in C source code (AFRL/Rome SBIR Phase-I and Phase-II projects “Source Code Vulnerability Analysis”)
3	Object Code Rewriting	Binary code (x86) rewriting technology for security (Jha, Miller, Giffin, and Christodorescu)	Java byte code (Jimple) rewriting technology for security (NIST SBIR Phase-I and Phase-II projects “Inline Reference Monitors for Object Code”)
4	Model Checking	(a) Weighted pushdown systems (Reps, Jha) (b) Verification of properties of concurrent C programs (Clarke, Sagar)	Verification of path properties in C and C++ programs (DARPA SBIR Phase-I and Phase-II projects “Verification of Hierarchical Graph Structures”)
5	Virus Detection	Detection of viruses in binary (x86) code. (Jha, Christodorescu)	(none)

C Technical Objectives

Our primary goal was to create a powerful, flexible, and open toolkit for static program analysis that would support multiple programming languages, multiple computer platforms, multiple analysis algorithms, and multiple client applications. A successful design would maximize code reuse, i.e., minimize the amount of code duplication that would be required of any given user of the toolkit.

Our secondary goals were (a) to provide GrammaTech technology to Wisconsin and CMU in support of their MURI projects, (b) to transition research results **from** those projects back into GrammaTech, and (c) to support early adopters of the transitioned research results in the Government.

The project was designed to be a win for each of the three parties involved:

- **Universities** would get access to high-quality, supported technology, thereby freeing them to focus on basic research. This would minimize their humdrum engineering activities, and minimize disruptions involved in day-to-day support of early adopters.
- **GrammaTech** would get access to world-class researchers and their prototypes for new cutting-edge products, as well as feedback from early adopters to guide the development of those products.
- **The Government** would avoid funding wasteful duplicate work, and accelerate transitions of basic research.

D Work Items Planned

The work items for the project were (1) to design new APIs with which users could program their static analyses; (2) to design a new architecture for CodeSurfer; (3) to implement the new architecture and APIs; (4) to create two reference implementations of static analysis algorithms using the new architecture and APIs; and (5) to evaluate the reference implementations and the experience of the Wisconsin project to assess the success of the project.

More specifically, we anticipated doing the following work to meet our primary objective:

1. **Multi-lingual capabilities.** This work would involve
 - a. *Abstracting* language-specific functionality out of CodeSurfer *per se*.
 - b. *Reinstantiating* pre-existing versions of CodeSurfer using the factored language-independent services we had introduced.
 - c. *Implementing* new front ends for one or more other programming languages to demonstrate the generality of the new architecture.
2. **Builder modularization.** This work would involve
 - a. *Decomposing* CodeSurfer's monolithic builder into its constituent analysis components, and creating an open API for each such component. This would allow clients of the system to program their own analyses by writing their own code using the API in whatever phase ordering or iteration schemes were most suitable for their own applications.
 - b. *Implementing* new analyses making use of the newly exposed components to demonstrate the generality of the new architecture.
3. **Back-end extensions.** This work would involve
 - a. *Creating* additional open APIs in the back-end to support plug-in applications.
 - b. *Demonstrating* the use of those APIs by various plug-in applications.

We also anticipated the following work to meet our secondary objectives:

- **Support to MURIs.** Discussions with the MURI researchers to elicit their requirements, interactions during an iterative design cycle, delivery of new versions of software, and bug fixing in a timely manner.
- **Transition from MURIs.** Adaptation and adoption of MURI research results within GrammaTech.
- **Outreach.** Discussions with prospective early adopters, packaging and documenting results in a distributable form, delivery to early adopters, and support to them. Writing of co-authored papers.
- **Reporting.** Participation in bi-annual MURI reviews.

E Results

Our primary technical results were as follows:

1. Multi-lingual capabilities.

- a. We developed a front-end System Development Kit (SDK) for CodeSurfer that facilitates creation of alternative front ends for different programming languages. The SDK contains
 - i. Abstract datatypes that can be used by a front end to build and output the intermediate representations needed by the CodeSurfer builder.
 - ii. The notion of a Language Module that contains all language-specific code and data needed by CodeSurfer.
 - iii. Complete documentation.

The SDK supports:

- i. A language-independent abstract-syntax-tree (AST) framework. The AST representation can be made available for use in front ends, in the dependence-graph builder, and in the back-end scripting language.
 - ii. A language-independent control-flow graph (CFG) definition facility.
 - iii. An abstract datatype for source-position information.
 - iv. Abstract datatypes in support of pointer analysis (see PAM, below).
- b. We instantiated our pre-existing *ad hoc* versions of CodeSurfer for C/C++ and Intel x86 using the SDK.
 - c. We used the SDK to implement a new version of CodeSurfer for Jimple [1], a three-address version of Java byte codes. (This work was funded by our NIST SBIR contract.)

2. Builder modularization.

- a. We factored CodeSurfer's pre-existing pointer analysis code into a separate Pointer Analysis Module (PAM) that can be used independently of CodeSurfer. PAM consists of

- i. An SDK for creating the intermediate representations needed by the pointer analysis engine.
 - ii. The pointer analysis engine itself.
 - iii. An API, termed the Pointer Analysis Data Base (PADB), for accessing the points-to results that have been computed by the pointer analysis engine.
 - b. We had originally anticipated modularizing and exposing the individual analysis components of the CodeSurfer builder, thereby allowing users to instantiate different “builders” of their own choosing. However, two concerns on the part of the PIs at Wisconsin, performance and intellectual property rights, led us to implement a quite different architecture. In short, Wisconsin wanted a light-weight analysis platform for x86 binaries that could be severed from CodeSurfer altogether. Accordingly, rather than modularizing the CodeSurfer builder, we were asked to provide an effective analysis infrastructure wholly within the x86 front end. In effect, the front-end SDK, which was intended just to facilitate a client’s access to the analysis capabilities of CodeSurfer’s builder, became the analysis platform itself. Some features of the CodeSurfer builder, e.g., basic block analysis, were lifted from the builder and replicated in the front end. Unfortunately, this was at odds with our goal of minimizing code duplication.
3. **Back-end extensions.** Joint work by GrammaTech and Wisconsin on buffer-overflow detection led to several back-end extensions:
- a. The creation of a general-purpose browser for viewing the results of code scans.
 - b. The creation of an open API for serializing CodeSurfer objects. This extension was needed to provide a persistent representation of the buffer-overflow results.

Our activities aimed at the secondary objectives were as follows:

- **Collaboration with the Wisconsin MURI.** In two of the application areas, Object Code Analysis and Buffer Overflow Analysis, the efforts at GrammaTech and Wisconsin became so tightly intertwined that it is not appropriate to describe those activities in simple uni-directional terms as “support to” or “transition from” the MURI. The efforts became true collaborations.
 - **Object Code Analysis.** Wisconsin and GrammaTech collaborated on the development of x86fe (a.k.a. “the connector”), which can be used as a standalone x86 analysis module, or as a CodeSurfer front end. Roughly speaking, the division of labor is that Reps and Balakrishnan work on Value Set Analysis (VSA), Affine Relation Analysis (ARA), and Aggregate Structure Identification (ASI) [4, 5], and GrammaTech does the rest including¹
 - i. *Use/Def Information.* Detailed use/kill/conditional-kill information is provided for every instruction.

¹ Note that some of the x86fe services listed were implemented in a second, follow-on contract the year after this contract ended. We provide the complete list in the interest of being clear about the services offered by x86fe today, we do not believe that it is essential to detail the exact year in which each service was implemented or further perfected.

- ii. *Register Live-Range Analysis (RLRA)*. The live-ranges of registers are computed.
 - iii. *Basic Blocks*. Basic blocks for the entire application (including libraries) are computed.
 - iv. *Call Graph*. The call supergraph for the entire program (including libraries) is computed.
 - v. *Support for Libraries*. A repository of pre-processed libraries is computed, from which individual procedures are demand loaded.
 - vi. *Spill Regions*. Regions of code are computed in which registers are "spilled" into memory locations (e.g., "mem = eax; ...; eax = mem;").
 - vii. *Register Save/Restore Instruction Pairs*. Pairs of instructions that are used to save/restore registers at a call (caller), or on entry to/exit from a procedure (callee), are computed.
 - viii. *Port Analysis*. Pseudo-variables are created for all ports accessed by the program. Port references are determined using constant propagation.
 - ix. *Support for Multiple-Entry-Point Functions*. Multiple-entry-point functions are detected and represented.
 - x. *Support for Clones*. Multiple-entry-point functions are optionally cloned and converted to single-entry-point functions.
 - xi. *Support for Non-linear Functions*. Instructions that are not included in any function(s) by IDAPro are added to the function(s) that end up executing them.
 - xii. *Support for Import Tables*. Functions and DLLs that are imported by the program are detected.
 - xiii. *Register Aliases*. A map of register aliases is maintained (e.g., al -> ax -> eax).
 - xiv. *End-to-end Connectivity with CodeSurfer*. x86fe and CodeSurfer are kept in synch.
- **Buffer Overrun Analysis**. Wisconsin and GrammaTech collaborated on the development of a tool for the detection of buffer-overrun vulnerabilities in ANSI C programs. During the year, we co-authored a paper describing our joint work [6]. We tested the tool on the current version (2.6.2) of the Washington University FTP daemon, a popular file transfer server, found 14 previously unreported overruns, and reported them to the developers.
- **Support to the MURIs.**
 - **PAM**. On 9/26/02, Prof. Jha requested that GrammaTech repackaging CodeSurfer's pointer analysis module as a stand alone component (PAM) for use in a joint CMU/Wisconsin project involving Prof. Clarke and his student Sagar Chaki. The user manual was delivered to CMU and Wisconsin on 12/20/02 (in Q3), although

delivery of the completed software did not occur for another month (in Q4). Creating PAM involved interacting closely with Mr. Chaki during the requirements, design, implementation, and deployment phases of the effort.

- **Transition from MURIs.**
 - **Model Checking.** We transitioned the work of Reps, Schwoon, and Jha on weighted pushdown systems [8], and their prototype implementation (Weighted Moped) to GrammaTech, where we used it as a model checking engine for the Path Inspector [7], a tool that checks sequencing properties in programs. The Path Inspector has been released as a commercial product.
- **Outreach.** In a separately funded effort, we worked to transition Wisconsin's research to SSC-SD (SPAWAR). As a first step, we trained a SPAWAR employee to use CodeSurfer and the prototype Wisconsin/GrammaTech buffer-overflow vulnerability detector. We then used the buffer-overflow tool to analyze the GCCS-M Tactical Management Service (TMS), and found one possible overrun in this fielded program, albeit not an overrun that can be exploited to seize control of the program.
- **Reporting.** We attended the semi-annual MURI review in Pittsburgh, and the semi-annual MURI review in Williamsburg. Tim Teitelbaum reported on GrammaTech's activities at both reviews.

F Conclusions

The project confirmed the hypothesis that it is technically possible to build an effective common infrastructure for the static analysis of software that meets the needs of a disparate collection of applications.

References

1. *Soot: a Java Optimization Framework*. McGill University.
2. *CodeSurfer User Guide and Reference Manual*. 2004, Ithaca, NY: GrammaTech, Inc.
3. Anderson, P. and T. Teitelbaum, *Software Inspection using CodeSurfer*. In *WISE 01, Workshop on Inspection in Software Engineering*. 2001. Paris.
4. Balakrishnan, G. and T. Reps, *Analyzing memory accesses in x86 binary executables*. 2003, Computer Sciences Department, University of Wisconsin, Madison, WI TR-1486.
5. Balakrishnan, G. and T. Reps, *Analyzing Memory Accesses in x86 Executables*. In *International Conference on Compiler Construction*. 2004.
6. Ganapathy, V., et al., *Buffer Overrun Detection using Linear Programming and Static Analysis*. In *10th ACM Conference on Computer and Communications Security*. 2003. Washington, DC.
7. GrammaTech, *The Path Inspector*. 2004, http://www.grammatech.com/products/codesurfer/overview_pi.html.
8. Reps, T., S. Schwoon, and S. Jha, *Weighted pushdown systems and their application to interprocedural dataflow analysis*. In *10th Int. Static Analysis Symp*. 2003. San Diego, CA.